Divide and conquer is a class of algorithmic techniques that work by:

① Breaking problem into smaller subproblems

② Recursively solving each subproblem

③ Combining answers from subproblems to obtain answer to initial problem

First Example: Mergesort

⇒ Algorithm for sorting a list of numbers.

High level idea:

Suppose we want to sort $A = \{10, 2, 5, 3, 7, 13, 1, 6\}$

Break into subproblems → 1 Split array in halves $\{10, 2, 5, 3\}$ and $\{7, 13, 1, 6\}$

Solve each subproblem → 2 Sort each half $\{2, 3, 5, 10\}$ and $\{1, 6, 7, 13\}$

Combine solutions → 3 Merge two sorted halves $\{1, 2, 3, 5, 6, 7, 10, 13\}$

How to implement Merge ?

⇒ Should take advantage of the fact that both B_L and B_R are sorted.

Idea: $B_L = \{2, 3, 5, 10\}$ $B_R = \{1, 6, 7, 13\}$

Pick smaller element between the two pointers and move the one that is chosen:

⇒ Pick 1 and move B_R pointer $B_L = \{2, 3, 5, 10\}$ $B_R = \{1, 6, 7, 13\}$

⇒ Pick 2 and move B_L pointer $B_L = \{2, 3, 5, 10\}$ $B_R = \{1, 6, 7, 13\}$

⇒ Pick 3 and move B_L pointer $B_L = \{2, 3, 5, 10\}$ $B_R = \{1, 6, 7, 13\}$

⇒ Pick 5 and move B_L pointer $B_L = \{2, 3, 5, 10\}$ $B_R = \{1, 6, 7, 13\}$

⇒ Pick 6 and move B_R pointer $B_L = \{2, 3, 5, 10\}$ $B_R = \{1, 6, 7, 13\}$

⇒ Pick 7 and move B_R pointer $B_L=\{2,3,5,10\}$ $B_R=\{1,6,7,13\}$

⇒ Pick 10 and move B_L pointer $B_L=\{2,3,5,10\}$ $B_R=\{1,6,7,13\}$

⇒ Pick 13 and move B_R pointer $B_L=\{2,3,5,10\}$ $B_R=\{1,6,7,13\}$

Merge $(X[0,\ldots,k-1], Y[0,\ldots,l-1])$

```
If X.length==1 return Y
If Y.length==1 return X
If X[0] ≤ Y[0]
    return {X[0], Merge(X[1,...,k-1], Y[0,...,l-1])}
If Y[0] > Y[0]
    return {Y[0], Merge(X[0,...,k-1], Y[1,...,l-1])}
```

Running Time $O(k+l)$

Correctness By induction on $k+l$. If $X[0] \le Y[0]$ and $\text{Merge}(X[1,\ldots,k-1], Y[0,\ldots,l-1])$ produces a sorted array then $\{X[0], \text{Merge}(X[1,\ldots,k-1], Y[0,\ldots,l-1])\}$ is sorted and similarly for the case $X[0]>Y[0]$

Mergesort $(A[0,\ldots,n-1])$

```
If n==1 return A
B_L = Mergesort(A[0,...,⌊n/2⌋-1])
B_R = Mergesort(A[⌊n/2⌋,...,n])
return Merge(B_L, B_R)
```

Correctness By induction of n. Suppose mergesort is correct on arrays of size $0,\ldots,n-1$. Then, the recursive calls produce sorted arrays, and the correctness of mergesort follows from the correctness of the Merge procedure.

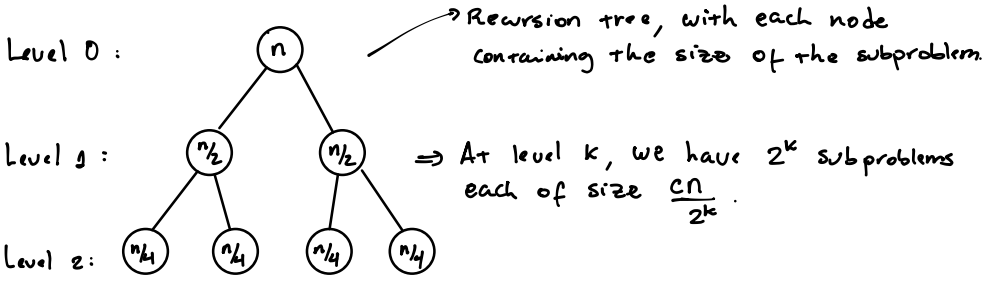
Running Time Merge requires $O(k+l)$ time.

$T(n)$ = running time of mergesort on an array of size n. Then:

$$T(n) = 2T(n/2) + O(n)$$

This a recurrence that can be solved by **unrolling** it:

$T(n) \leq 2 \cdot T(n/2) + cn$ for a suitable constant $c > 0$.



Hence, total work is:

$$cn + 2 \cdot \frac{cn}{2} + 4 \cdot \frac{cn}{4} + \dots + \frac{cn}{2^h} \cdot 2^h = \sum_{k=0}^{h} \frac{cn}{2^k} \cdot 2^k = c \cdot n \cdot h$$

⇒ h is the depth of tree. In this case $h = O(\log n)$, so

$$T(n) = O(n \log n)$$

Can we do better? Recall that this is the best we can do for sorting via comparisons.

Note: Analyzing the running time of Divide and Conquer algorithms, typically involves analyzing recurrences

In particular, we obtain recurrences of the form:

$$T(n) = a \cdot T\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$$

- $O(n^d)$ is the cost of combining the solutions of subproblems
- a is the # of subproblems
- $\left\lceil \frac{n}{b} \right\rceil$ is the size of each subproblem

Master Theorem

If $T(n) = a \cdot T(\lceil \frac{n}{b} \rceil) + O(n^d)$ for some $a > 0$, $b > 1$ and $d \geq 0$,

then:

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Example 1: $T(n) = 2T(n/2) + O(n)$ (Mergesort)

$a = 2, b = 2, d = 1$

$\log_b a = 1$

$T(n) = O(n \log n)$.

Example 2: $T(n) = 3T(n/2) + O(n)$

$a = 3 \quad b = 2 \quad d = 1$

$\log_2 3 \approx 1.59$

$T(n) = O(n^{1.59})$

Notes

1) There are other variants of the Master Theorem (some are more general) but this one has a good balance between simplicity and power.

2) If we want to be completely precise, the running time of mergesort is

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n),$$

but for all the recurrences we will encounter (and most that arise in practice) we can ignore the ceiling and floor. Note that we may assume that n is a power of 2 (or more generally of b) and since $b^k \leq n \leq b^{k+1}$ for some k, the order of $T(n)$ is not affected.

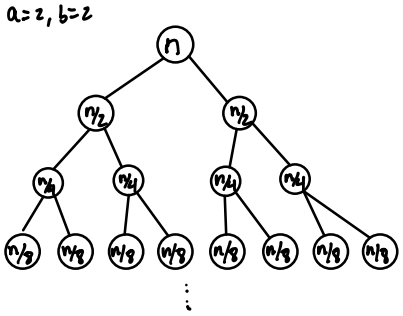
Proof of Master Theorem

We assume without loss of generality that n is a power of b.

(Otherwise $b^k < n < b^{k+1}$ and we do the analysis for $b^{k+1} \leq n \cdot b$.)

Idea Use the recursion tree as when unrolling.

$a=2, b=2$



$\Rightarrow$ Size of subproblem at level k is $\frac{n}{b^k}$

$\Rightarrow$ Number of subproblems at level k is a^k

$\Rightarrow$ Work done to solve subproblem at level k:

$$O\left(\left(\frac{n}{b^k}\right)^d\right)$$

$\Rightarrow$ Height of tree is $h = \log_b n$

So, overall we get:

$$a^0 \cdot O\left(\left(\frac{n}{b^0}\right)^d\right) + a^1 O\left(\left(\frac{n}{b^1}\right)^d\right) + a^2 \cdot O\left(\left(\frac{n}{b^2}\right)^d\right) + \dots + a^h \cdot O\left(\left(\frac{n}{b^h}\right)^d\right)$$

$$= \sum_{k=0}^{h} a^k \cdot O\left(\left(\frac{n}{b^k}\right)^d\right) = O(n^d) \cdot \sum_{k=0}^{h} \left(\frac{a}{b^d}\right)^k = S$$

Now we use Problem 3 from HW 1

$\Rightarrow$ If $a = b^d$ (or $\log_b a = d$), $S = O(n^d \cdot h) = O(n^d \cdot \log n)$

$\Rightarrow$ If $a < b^d$ (or $\log_b a < d$), $S = O(n^d)$

$\Rightarrow$ If $a > b^d$ (or $\log_b a > d$), $S = O(n^d) \cdot \left(\frac{a}{b^d}\right)^h = O(n^d) \cdot \frac{a^{\log_b n}}{b^{d \cdot \log_b n}}$

$$= \frac{O(n^d) \cdot a^{\log_a n / \log_a b}}{n^d}$$

$$= O\left(n^{1/\log_a b}\right) = O\left(n^{\log_b a}\right),$$

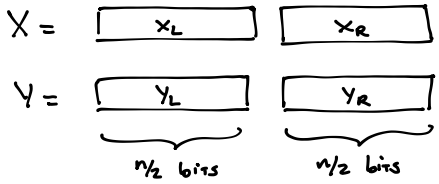
and the proof is complete. ∎

Second Divide & Conquer Example — Integer Multiplication

Input Two n bit numbers X and Y

Output $X \cdot Y$

Idea (Goes back to Gauss)



Example: $X = 1011\,0110$

$X_L = 1011$

$X_R = 0110$

Fact. $X = X_L \cdot 2^{n/2} + X_R$

Proof. $X_L \cdot 2 = X_L 0$ (recall that multiplying by 2 corresponds to a shift)

So,

$$X_L \cdot 2^{n/2} = X_L \underbrace{0 \ldots 0}_{n/2 \text{ zeros.}}$$

and

$$X_L \cdot 2^{n/2} + X_R = X$$ ∎

Then,

$$X \cdot Y = (X_L \cdot 2^{n/2} + X_R)(Y_L \cdot 2^{n/2} + Y_R)$$

$$= X_L \cdot Y_L \cdot 2^n + X_L \cdot Y_R \cdot 2^{n/2} + X_R Y_L \cdot 2^{n/2} + X_R Y_R$$

⇒ So, to compute $X \cdot Y$, it suffices to compute $X_L Y_L$, $X_L Y_R$, $X_R Y_L$ and $X_R Y_R$.

⇒ If we had these four values, computing $X \cdot Y$ can be done in $O(n)$, since it involves sums and multiplications by 2 (shifts).

⇒ Therefore, if $T(n)$ is the running time of this algorithm, we have:

$$T(n) = 4 \cdot T(n/2) + O(n).$$

$a = 4 \quad b = 2 \quad d = 1 \quad \log_2 4 = 2$

and by Master Theorem, $T(n) = O(n^2)$ → Not better than naive method!

We can improve by noting that:

$$X_R Y_L + X_L Y_R = (X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R$$

So:

$$X \cdot Y = X_L \cdot Y_L \cdot 2^n + (X_L \cdot Y_R + X_L Y_R) 2^{n/2} + X_R Y_R$$

$$= X_L \cdot Y_L \cdot 2^n + \left[(X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R \right] \cdot 2^{n/2} + X_R Y_R.$$

and we only need to compute $X_L Y_L$, $X_R Y_R$ and $(X_L + X_R)(Y_L + Y_R)$.

So, we save one multiplication. Is this better?

$$T(n) = 3\,T(n/2) + O(n)$$

$a = 3 \quad b = 2 \quad \log_2 3 \approx 1.59 \quad d = 1$

$T(n) = O(n^{1.59})$ by the Master Theorem.

Much better than naive algorithm!

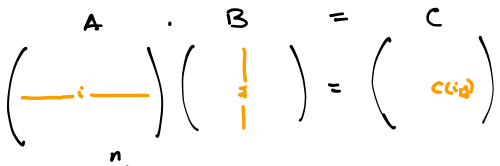
Third Divide & Conquer example

Strassen's Algorithm for Matrix Multiplication

Input Two $n \times n$ matrices A and B

Output $C = A \cdot B$

Recall:



$$C(i,j) = \sum_{k=1}^{n} A(i,k) \cdot B(k,j)$$

$\Rightarrow$ Then we can compute C in $O(n^3)$ time.

$\Rightarrow$ Can we do better? Reading A and B takes $\Theta(n^2)$ time, so there is room for improvement.

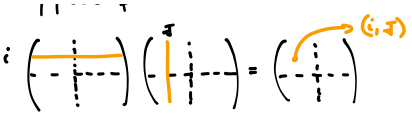
Breaking into subproblems: Perform matrix multiplication blockwise.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

where A_{ij} and B_{ij} are $n/2 \times n/2$ matrices.

Fact $C = A \cdot B = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$

Proof. Suppose first that $i < n/2$ and $j = n/2$. That is



Then, $C(i,j) = \sum_{k=1}^{n} A(i,k) \cdot B(k,j)$

$$= \sum_{k=1}^{n/2} A(i,k) \cdot B(k,j) + \sum_{k=\frac{n}{2}+1}^{n} A(i,k) \cdot B(k,j)$$

$$= \sum_{k=1}^{n/2} A_{11}(i,k) \cdot B_{11}(k,j) + \sum_{k=0}^{n/2} A_{12}(i,k) \cdot B_{21}(k,j)$$

$$= A_{11} \cdot B_{11}(i,j) + A_{12} \cdot B_{21}(i,j)$$

All other cases for i and j can be checked in the same manner which would complete the proof ∎

With this fact we obtain a Divide & Conquer algorithm with 8 subproblems of size $n/2$.

The time it takes to combine the subproblems is $O(n^2)$ (4 $n \times n$ matrix sums), so the running time satisfies:

$$T(n) = 8 \cdot T(n/2) + O(n^2)$$

$a = 8 \quad b = 2 \quad d = 2 \quad \log_2 8 = 3$

By Master Theorem: $T(n) = O(n^3)$.

↳ same as naive method!

Strassen's idea (1969)

Lemma Let

$$P_1 = A_{11}(B_{12} - B_{22}) \qquad P_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$P_2 = (A_{11} + A_{12}) B_{22} \qquad P_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$P_3 = (A_{21} + A_{22}) B_{11} \qquad P_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

$$P_4 = A_{22}(B_{21} - B_{11})$$

Then,

$$A \cdot B = C = \begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{pmatrix}$$

Proof.

$$P_1 + P_2 = A_{11}(B_{12} - B_{22}) + (A_{11} + A_{12})B_{22}$$

$$= A_{11} \cdot B_{12} - A_{11}B_{22} + A_{11}B_{22} + A_{12}B_{22}$$

$$= A_{11}B_{12} + A_{12}B_{22} \quad \checkmark$$

$$P_5 + P_4 - P_2 + P_6 = (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) - (A_{11} + A_{12})B_{22}$$

$$+ (A_{12} - A_{22})(B_{21} + B_{22})$$

$$= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} + A_{22}B_{21} - A_{22}B_{11} - A_{11}B_{22} - A_{12}B_{22}$$

$$+ A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22}$$

$$= A_{11}B_{11} + A_{12}B_{21} \quad \checkmark$$

$$P_3 + P_4 = (A_{21} + A_{22})B_{11} + A_{22}(B_{21} - B_{11})$$

$$= A_{21}B_{11} + A_{22}B_{11} + A_{22}B_{21} - A_{22}B_{11}$$

$$= A_{21}B_{11} + A_{22}B_{21} \quad \checkmark$$

$$P_1 + P_5 - P_3 - P_7 = A_{11}(B_{12} - B_{22}) + (A_{11} + A_{22})(B_{11} + B_{22}) - (A_{21} + A_{22})B_{11}$$

$$- (A_{11} - A_{21})(B_{11} + B_{12})$$

$$= A_{11}B_{12} - A_{11}B_{22} + A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} - A_{21}B_{11} - A_{22}B_{11}$$
$$- A_{11}B_{11} - A_{11}B_{12} + A_{21}B_{11} + A_{21}B_{12}$$

$$= A_{21}B_{12} + A_{22}B_{22} \quad \checkmark$$

■

Now, computing $P_1, \ldots, P_7$ requires 7 $\frac{n}{2} \times \frac{n}{2}$ matrix multiplications

Therefore:

$$T(n) = 7 \cdot T(n/2) + O(n^2)$$

$$a = 7 \quad b = 2 \quad d = 2 \qquad \log_2 7 = 2.81$$

So, $T(n) = O(n^{2.81})$ by Master Theorem.

Can we do better?

* Coppersmith – Winograd Algorithm (1990)

$$O(n^{2.376})$$

* Improved to $O(n^{2.373})$ in 2013 and to $O(n^{2.37285})$ in 2021!

These algorithms are not used in practice, since they are **galactic algorithms**, where the constant hidden by the O notation is so large, that for them to be better, n has to be so large that they would never be used on Earth.

Fourth Divide & Conquer Example: Peak Finding

Given an array $A[0, \ldots, n-1]$, we say that $A[i]$ is a peak if it is not smaller than its neighbor(s):

$$A[i-1] \geq A[i] \geq A[i+1]$$

where we imagine that $A[-1] = A[n] = \infty$.

Example:

5	4	3	1	2	6	7	4

Only 5 and 7 are peaks.

Goal Find **any** peak.

Brute-Force algorithm: $O(n)$ running time.

Can we do better? Yes!

Consider any element $A[i]$ $(0 < i < n-1)$. There are four options:

① $A[i-1] \leq A[i]$ and $A[i] \geq A[i+1]$ 

– Peak

② $A[i-1] \leq A[i] \leq A[i+1]$ 

– Increasing

③ $A[i-1] \geq A[i] \geq A[i+1]$ 

– Decreasing

④ $A[i-1] \geq A[i]$ and $A[i] \leq A[i+1]$ 

– Valley.

Idea Pick i. If $A[i]$ is a peak, we are done.

Otherwise, if $A[i] \leq A[i+1]$, there must be a peak in $[i+1, \ldots, n-1]$

If $A[i] \geq A[i-1]$, there must be a pick in $[0, \ldots, i-1]$

For the algorithm, we pick i to minimize the size of larger part, so we take $i = \lfloor \frac{n}{2} \rfloor$ the middle element.

Peak (A , i, J)

Input Array A with n numbers; indexes i and J between which we want to find peak. (initially i=0, J=n-1)

Output position of peak

$m = \lfloor \frac{i+J}{2} \rfloor$

If $A[m-1] \leq A[m]$ and $A[m] \geq A[m+1]$
  return m;

Else if $A[m] < A[m-1]$
  return Peak (A, i, m-1)

Else if $A[m] < A[m+1]$
  return Peak (A, m+1, J)

Note: We are assuming that $A[0] = A[n] = -\infty$; otherwise need to add boundary checks!

Running Time

$$T(n) = T(n/2) + O(1)$$

$a = 1 \quad b = 2 \quad d = 0 \quad \log_2 1 = 0$

$T(n) = O(\log n)$ by Master Theorem